Multipliers, Algorithms, and Hardware Designs

Mahzad Azarmehr

Supervisor: Dr. M. Ahmadi

Spring 2008

Outline

- Survey Objectives
- Basic Multiplication Schemes
  - Shift/Add Multiplication Algorithm
  - Basic Hardware Multiplier
- High-Radix Multipliers
  - Multiplication of Signed Numbers
  - Radix-4 Multiplication
  - Modified Booth's Recoding
- Tree and Array Multipliers
  - Using Carry-save Adders
  - Full Tree Multipliers
  - Alternative Reduction Trees
  - Tree Multipliers for signed numbers
- Variation in Multipliers
  - Divide and Conquer Design
  - Additive Multiply Modules
  - Bit-Serial Multipliers
  - Modular Multipliers
  - Squaring
- Conclusion

Survey Objectives

- Multiplication is a heavily used arithmetic operation that figures prominently in signal processing and scientific applications
- Multiplication is hardware intensive, and the main criteria of interest are higher speed, lower cost, and less VLSI area
- The main concern in classic multiplication, often realized by K cycles of shifting and adding, is to speed up the underlying multi-operand addition of partial products
- In this survey, a variety of multiplication algorithms and hardware designs are discussed

Shift/Add Multiplication Algorithm

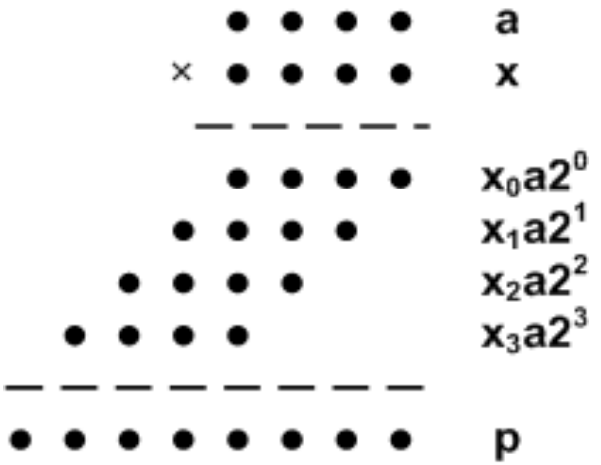
- With the following notation:

  a Multiplicand $a_{k-1}a_{k-2}\ldots a_1a_0$

  x Multiplier $x_{k-1}x_{k-2}\ldots x_1x_0$

  p Product $p_{2k-1}p_{2k-2}\ldots p_1p_0$

  Each row corresponds to the product of the multiplicand and a single bit of multiplier. Each term is either 0 or a



- Binary multiplication reduces to adding a set of numbers, each of which is 0, or shifted version of the multiplicand a

Shift/Add Multiplication Algorithm

- Sequential multiplication can be done by a cumulative partial product (initialized to 0) and successively adding to it the properly shifted terms $x_j a$

$$p^{(j+1)} = (p^{(j)} + x_j a 2^k)\, 2^{-l}$$

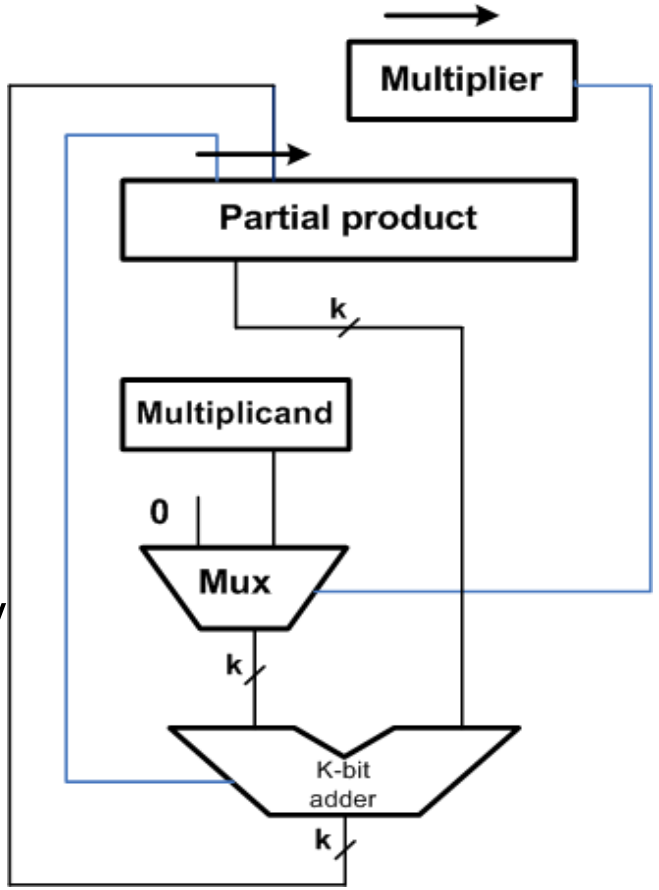
- Instead of shifting successive numbers to the left for alignment, cumulative partial product is shifted by one bit to the right

- The product will have a total shift of k bits to the right, so we pre-multiply a by 2^k to offset this effect

a	1 0 1 0
x	1 0 1 1
$P^{(0)}$	0 0 0 0
$+ x_0 a$	1 0 1 0
$2P^{(1)}$	0 1 0 1 0
$P^{(1)}$	0 1 0 1 0
$+ x_1 a$	1 0 1 0
$2P^{(2)}$	0 1 1 1 1 0
$P^{(2)}$	0 1 1 1 1 0
$+ x_2 a$	0 0 0 0
$2P^{(3)}$	0 0 1 1 1 1 0
$P^{(3)}$	0 0 1 1 1 1 0
$+ x_3 a$	1 0 1 0
$2P^{(4)}$	0 1 1 0 1 1 1 0
$P^{(4)}$	0 1 1 0 1 1 1 0

Basic Hardware Multiplier

- x and p are stored in shift registers
- The next bit of x is used to select 0 or a for addition
- Shifting can be performed by connecting the (i)th sum output to the (k+i-1)th bit of the partial product register and the adder's carry out to bit 2k-1
- x and lower half of p can share the same register



Multiplication of Signed Numbers

- In signed-magnitude numbers, the product's sign should be computed separately by XORing the operand signs
- In 2's-complement representation:
  - Negative multiplicand, the same routine with sign-extended values
  - Negative multiplier, the term $x_{k-1}a$ should be subtracted rather than added in the last cycle
- In practice, the required subtraction is performed by adding the 2's-complement of the multiplicand or adding its 1's-complement and inserting a carry-in of 1 into the adder

Multiplication of Signed Numbers

- Examples of 2's-complement multiplications:

a	1 0 1 1 0
x	0 1 0 1 1
$P^{(0)}$	0 0 0 0 0
$+ x_0 a$	1 0 1 1 0
$2P^{(1)}$	1 1 0 1 1 0
$P^{(1)}$	1 1 0 1 1 0
$+ x_1 a$	1 0 1 1 0
$2P^{(2)}$	1 1 0 0 0 1 0
$P^{(2)}$	1 1 0 0 0 1 0
$+ x_2 a$	0 0 0 0 0
$2P^{(3)}$	1 1 1 0 0 0 1 0
$P^{(3)}$	1 1 1 0 0 0 1 0
$+ x_3 a$	1 0 1 1 0
$2P^{(4)}$	1 1 0 0 1 0 0 1 0
$P^{(4)}$	1 1 0 0 1 0 0 1 0
$+ x_4 a$	0 0 0 0 0
$2P^{(5)}$	1 1 1 0 0 1 0 0 1 0
$P^{(5)}$	1 1 1 0 0 1 0 0 1 0

a	1 0 1 1 0
x	1 0 1 0 1
$P^{(0)}$	0 0 0 0 0
$+ x_0 a$	1 0 1 1 0
$2P^{(1)}$	1 1 0 1 1 0
$P^{(1)}$	1 1 0 1 1 0
$+ x_1 a$	0 0 0 0 0
$2P^{(2)}$	1 1 1 0 1 1 0
$P^{(2)}$	1 1 1 0 1 1 0
$+ x_2 a$	1 0 1 1 0
$2P^{(3)}$	1 1 0 0 1 1 1 0
$P^{(3)}$	1 1 0 0 1 1 1 0
$+ x_3 a$	0 0 0 0 0
$2P^{(4)}$	1 1 1 0 0 1 1 1 0
$P^{(4)}$	1 1 1 0 0 1 1 1 0
$+ (-x_4 a)$	0 1 0 1 0
$2P^{(5)}$	0 0 0 1 1 0 1 1 1 0
$P^{(5)}$	0 0 0 1 1 0 1 1 1 0

Multiplication using Booth's Recoding

- The more 1s there are in x, the slower the multiplication
- In Booth's recoding, every sequence of 1s is replaced with a sequence of 0s, a -1 in the least significant end, and addition of 1 in the next higher position:

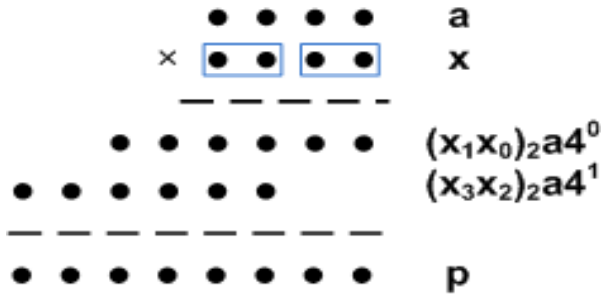
$$2^j+2^{j-1}+\ldots+2^{i+1}+2^i = 2^{j+1}-2^i$$

x_i	x_{i-1}	y_i	explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s
1	0	-1	Beginning of string of 1s
1	1	0	Continuation of string of 1s

a	1 0 1 1 0
x	1 0 1 1 1
y	-1 1 0 0 -1
$P^{(0)}$	0 0 0 0 0
+ y_0a	0 1 0 1 0
$2P^{(1)}$	0 1 0 1 0
$P^{(1)}$	0 0 1 0 1 0
+ y_1a	0 0 0 0 0
$2P^{(2)}$	0 0 1 0 1 0
$P^{(2)}$	0 0 0 1 0 1 0
+ y_2a	0 0 0 0 0
$2P^{(3)}$	0 0 0 1 0 1 0
$P^{(3)}$	0 0 0 0 1 0 1 0
+ y_3a	1 0 1 1 0
$2P^{(4)}$	1 1 0 1 1 1 0 1 0
$P^{(4)}$	1 1 0 1 1 1 0 1 0
+ y_4a	0 1 0 1 0
$2P^{(5)}$	0 0 1 0 1 1 0 1 0
$P^{(5)}$	0 0 0 1 0 1 1 0 1 0

High-Radix Multipliers

- These multiplication schemes handle more than one bit of the multiplier in each cycle

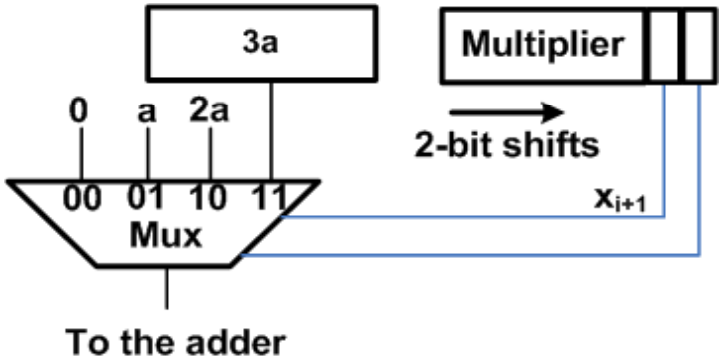


- A higher representation radix leads to fewer digits. Thus, a digit-at-a-time multiplication algorithm requires fewer cycles as we move to higher radices, which means fewer partial products

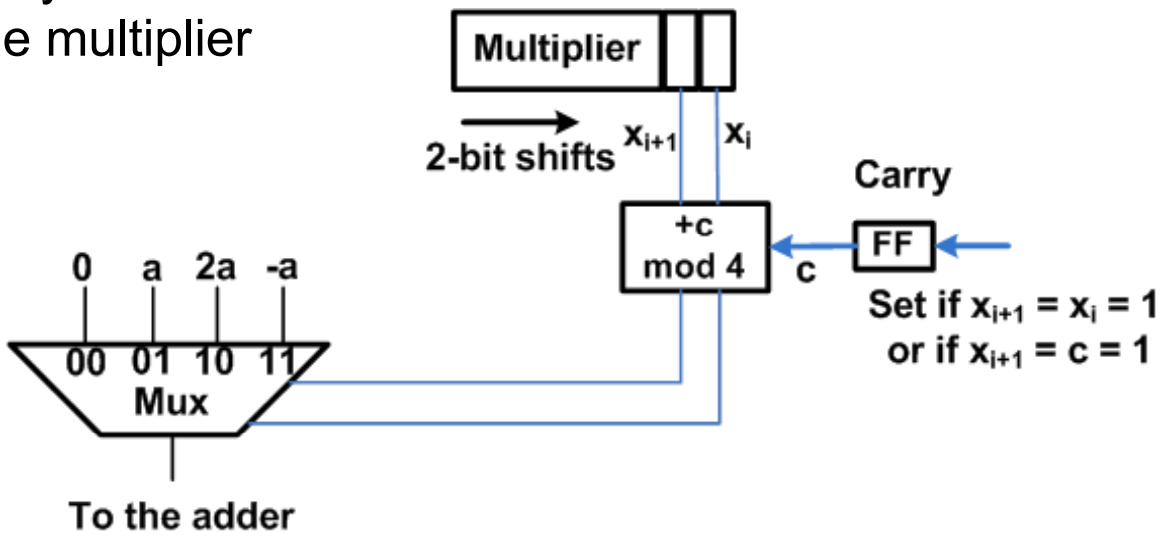
- The reduction in the number of cycles, along with the use of recoding and carry-save adders, leads to significant gains in speed over basic multipliers

Radix-4 Multipliers

- Based on two least significant end bits of multiplier, a pre-computed multiple of a is added



- Alternately, rather than adding 3a, add –a and send a carry of 1 into the next radix-4 digit of the multiplier



Modified Booth's Recoding

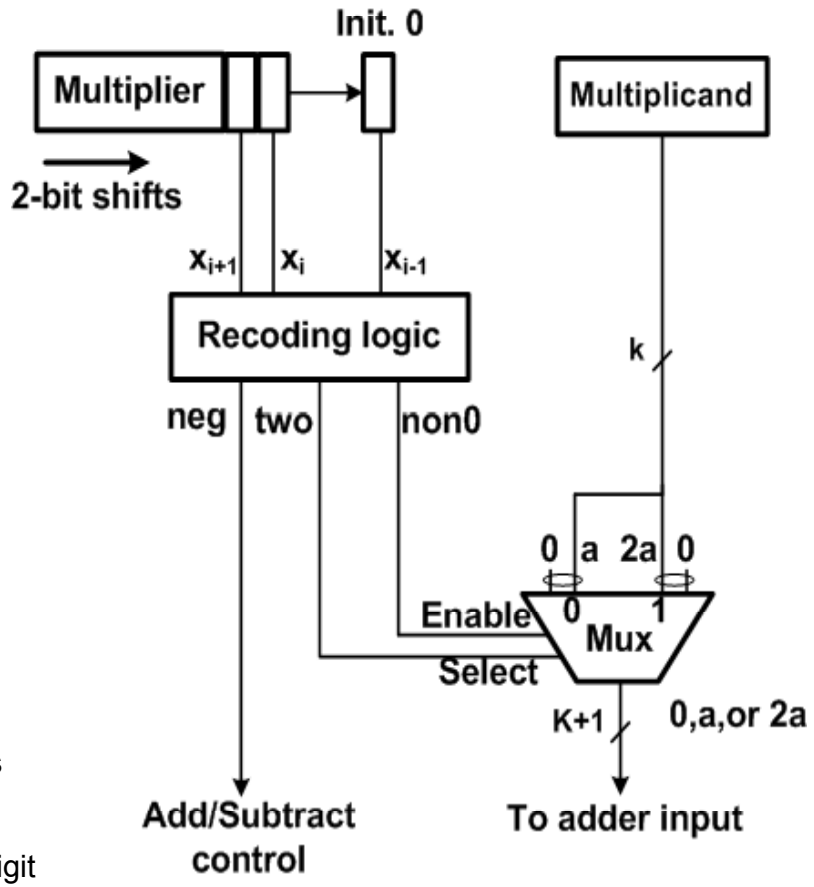
- If radix-4 multiplication is performed with the recoded multiplier, only the multiples of ±a and ±2a will be required, all of which are easily obtained by shifting and/or complementation

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	explanation
0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	End of a string of 1s
0	1	0	1	-1	Isolated 1 in x
0	1	1	1	0	End of a string of 1s
1	0	0	-1	0	Beginning of a string of 1s
1	0	1	-1	1	End one string, begin new string
1	1	0	0	-1	Beginning of a string of 1s
1	1	1	0	0	Continuation of string of 1s

Radix-4 Multipliers

- Booth's recoding is fully paralleled and carry-free

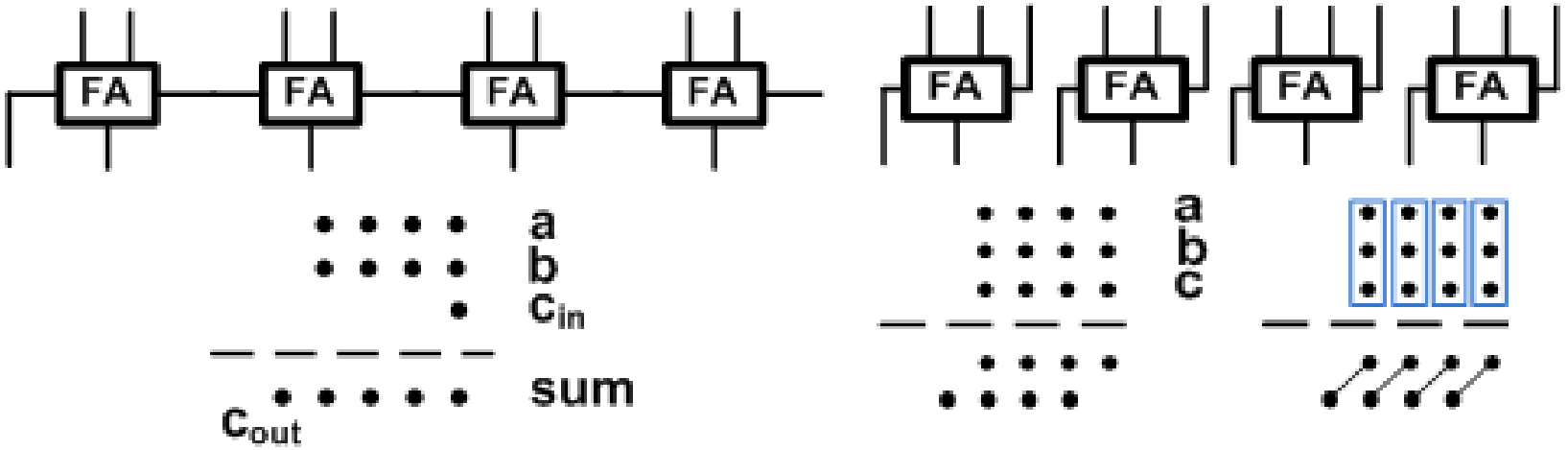
a	0 1 1 0	
x	1 0 1 0	Radix-4 recoded version of x
z	-1 -2	
$P^{(0)}$	0 0 0 0 0 0	
+ $z_0 a$	1 1 0 1 0 0	
$4P^{(1)}$	1 1 0 1 0 0	
$P^{(1)}$	1 1 1 1 0 1 0 0	
+ $z_1 a$	1 1 1 0 1 0	
$4P^{(2)}$	1 1 0 1 1 1 0 0	
$P^{(2)}$	1 1 0 1 1 1 0 0	



- non0: 1 bit to distinguish 0 from nonzero digits
- neg: 1 bit to show the sign of nonzero digit
- two: 1 bit to show the magnitude of nonzero digit

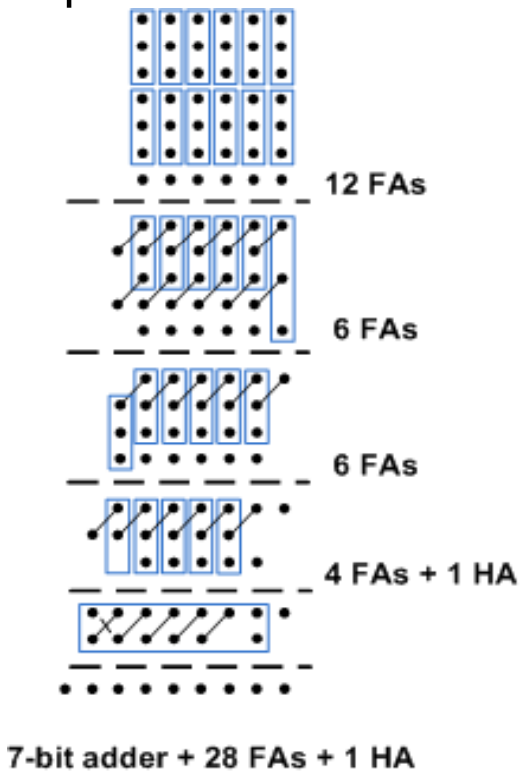
Using Carry-Save Adders

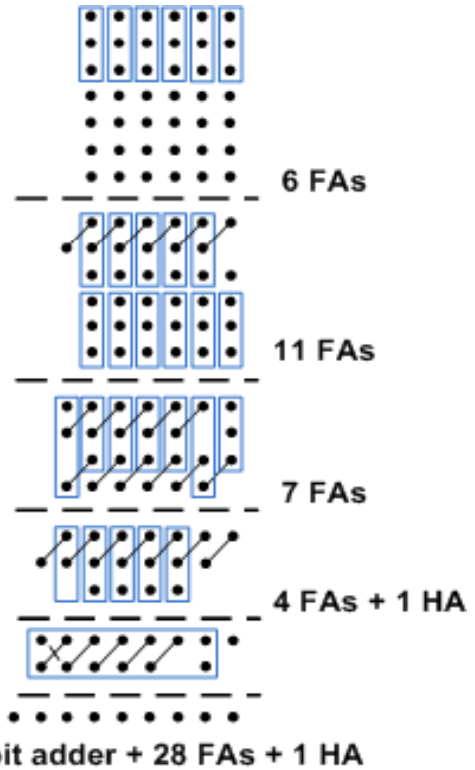
- Carry-save adders (CSA) can be used to reduce the number of addition cycles as well as to make each cycle faster
- A row of binary FA is used as a mechanism to reduce three numbers to two numbers, rather than finding a single “sum”



Wallace and Dadda trees

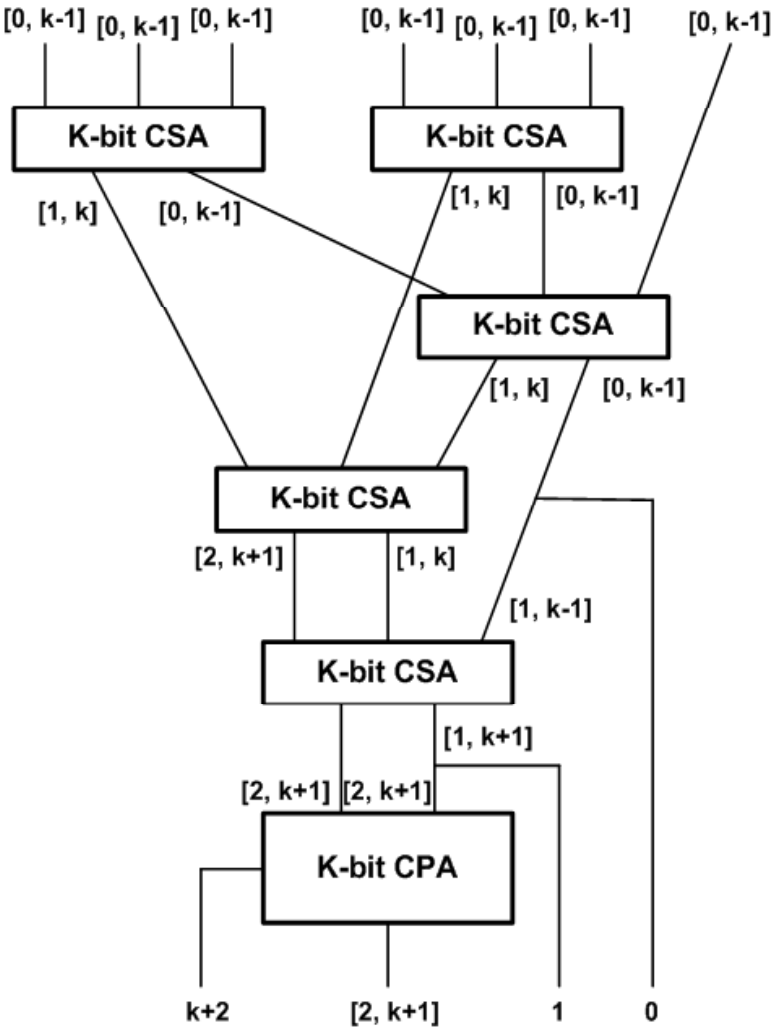
- Wallace's strategy is to combine the partial product bits at the earliest opportunity, which leads to the fastest possible design
- With Dadda's method, combining takes place as late as possible and usually leads to simpler CSA tree and a wider CPA





Using Carry-Save Adders

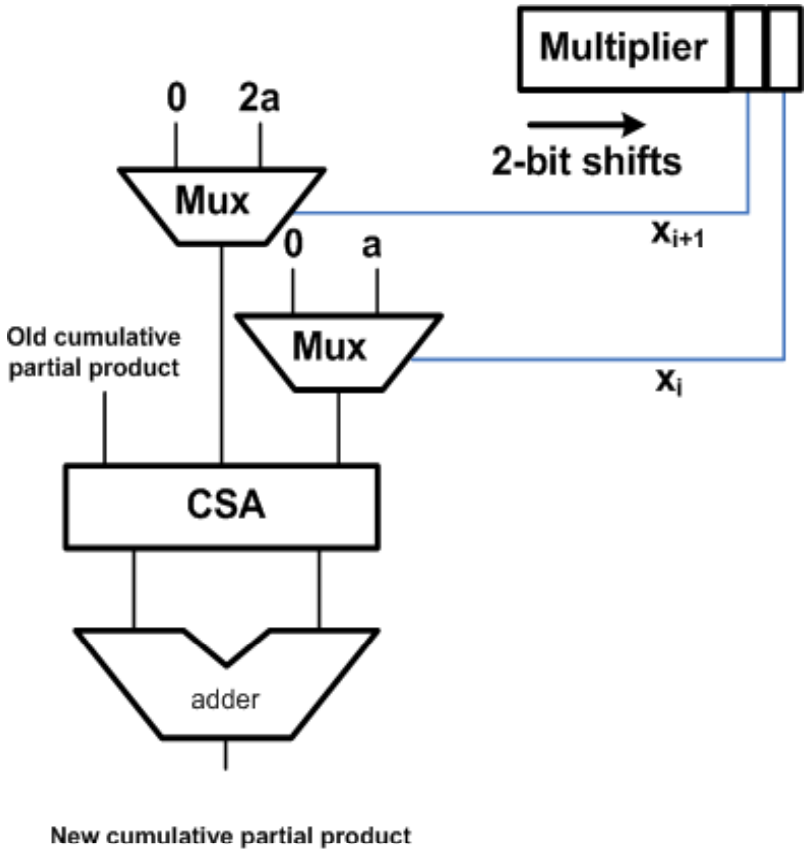
- A carry-save adder tree can reduce n binary numbers to two numbers having the same sum in O(log n) levels
- As an example, this CSA tree, reduces seven k-bit operands to two (k+2)-bit operands
- Not necessarily all the operands have the same alignment



Using Carry-Save Adders

- Radix-4 multiplication without Booth's recoding can be implemented by using a CSA to handle the 3a multiple

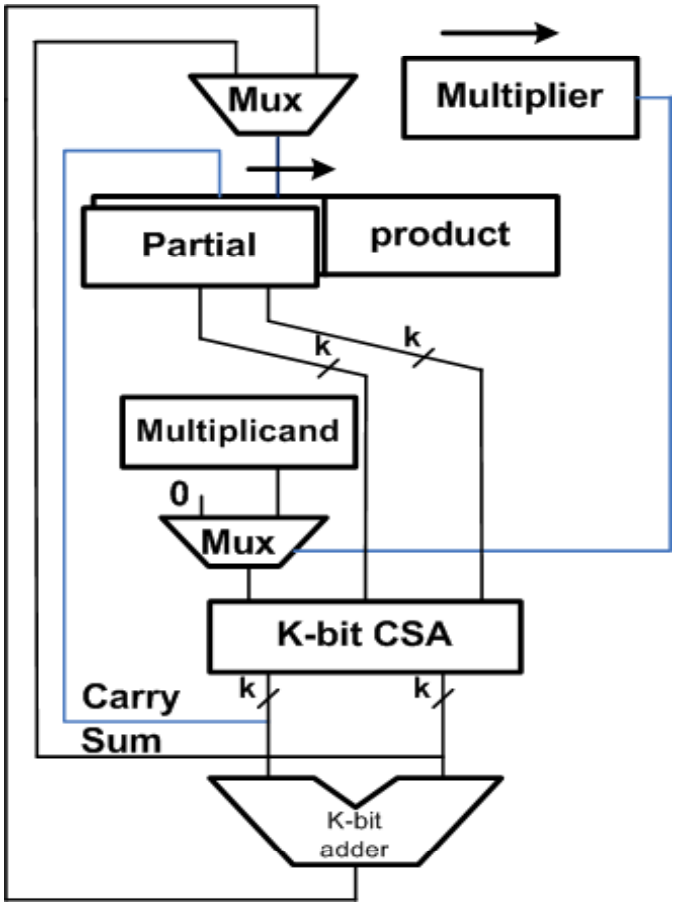
- The drawback is that the add time is slightly increased, since the CSA overhead is paid in every cycle, regardless of whether 3a is actually needed



Using Carry-Save Adders

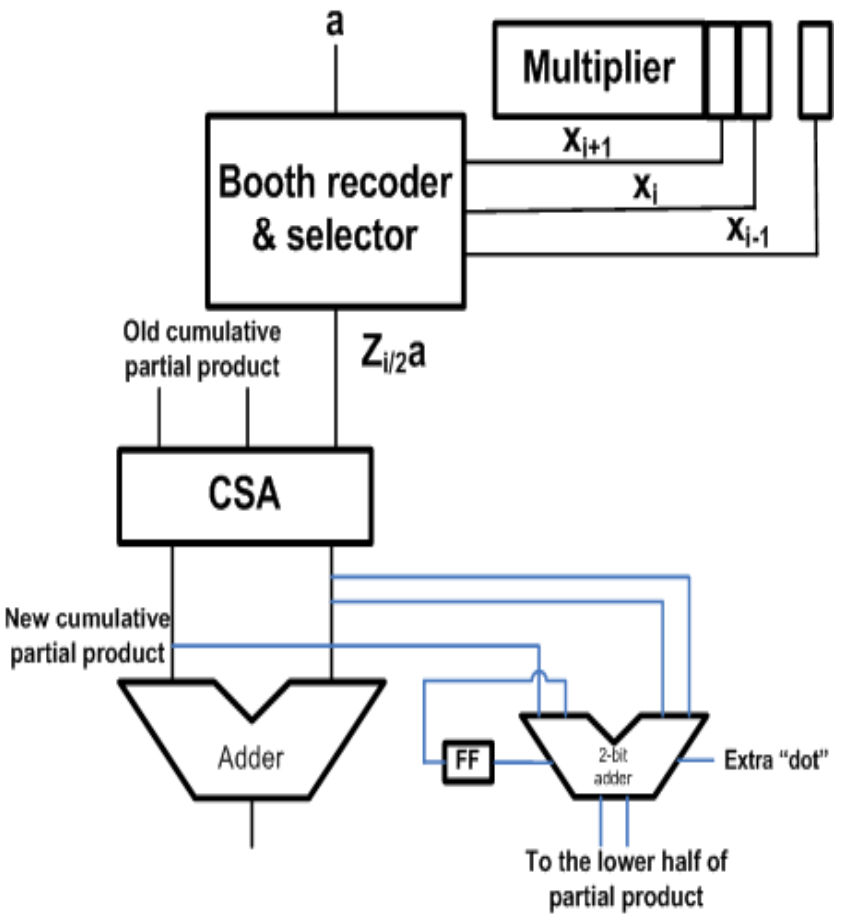
- CSA can be put to better use for reducing the addition time by keeping the cumulative partial product in stored-carry form

- As the three values that form the next cumulative partial product are added, one bit of the final product is obtained and shifted into the lower half of the register. This eliminates the need for carry propagation in all but the final addition



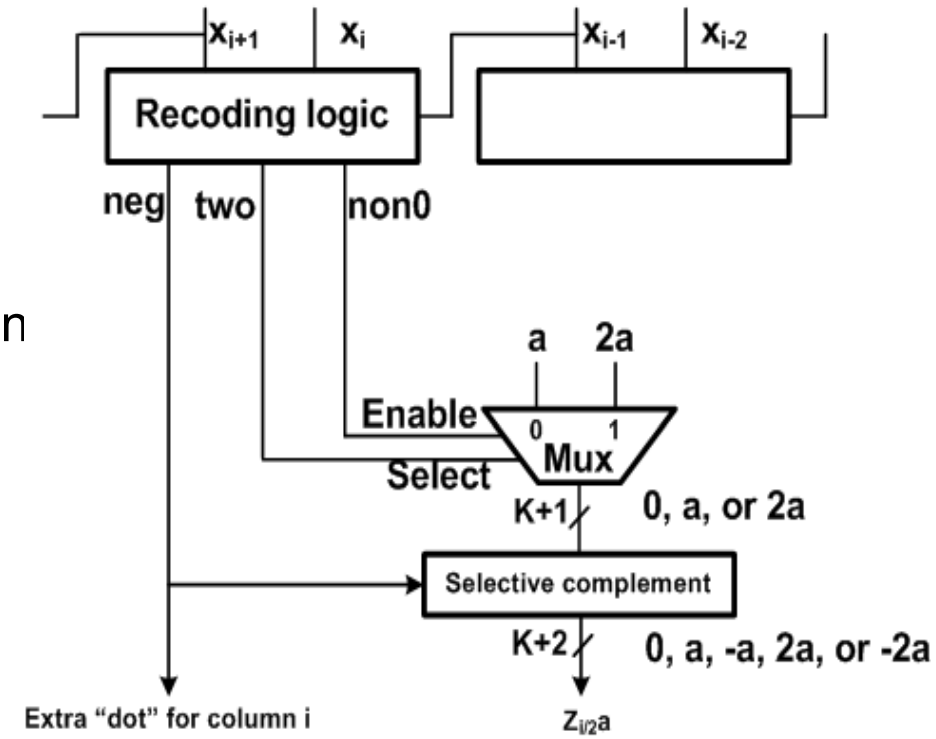
Using Carry-Save Adders

- The previous CSA-based design can be combined with radix-4 Booth's recoding to reduce the number of cycles by 50%, while also making each cycle considerably faster



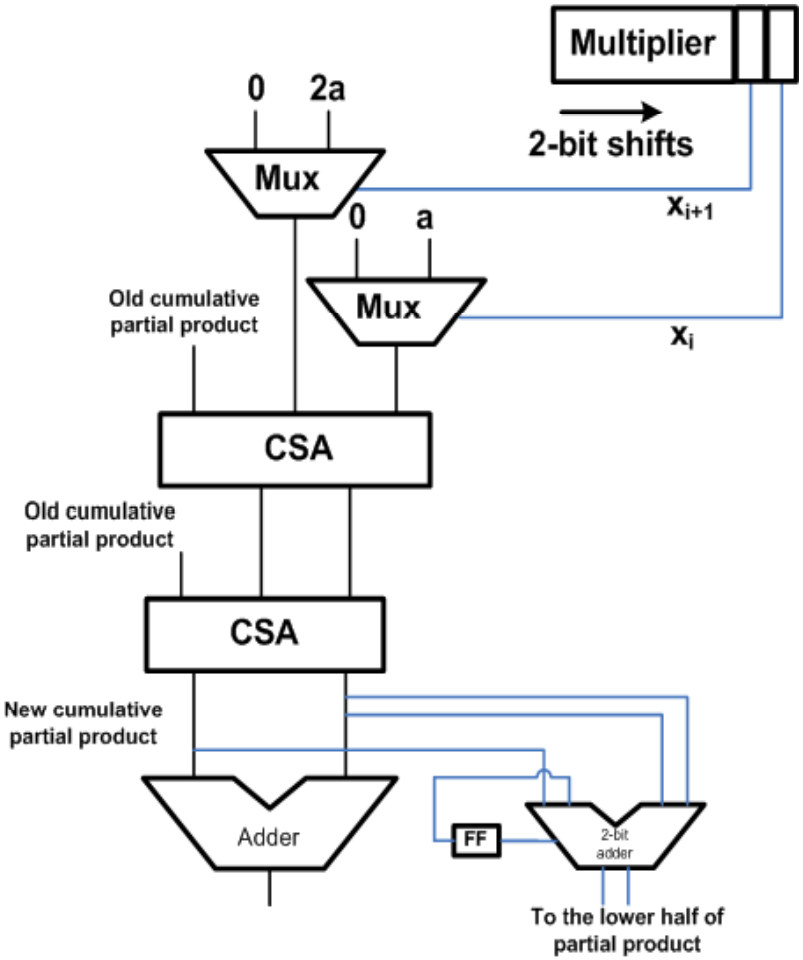
Using Carry-Save Adders

- In the Booth recoding logic and multiple selection circuit, the sign of each multiple must be incorporated in the multiple itself, rather than as a signal that controls addition/subtraction
- This configuration can be used for high-radix and parallel multipliers



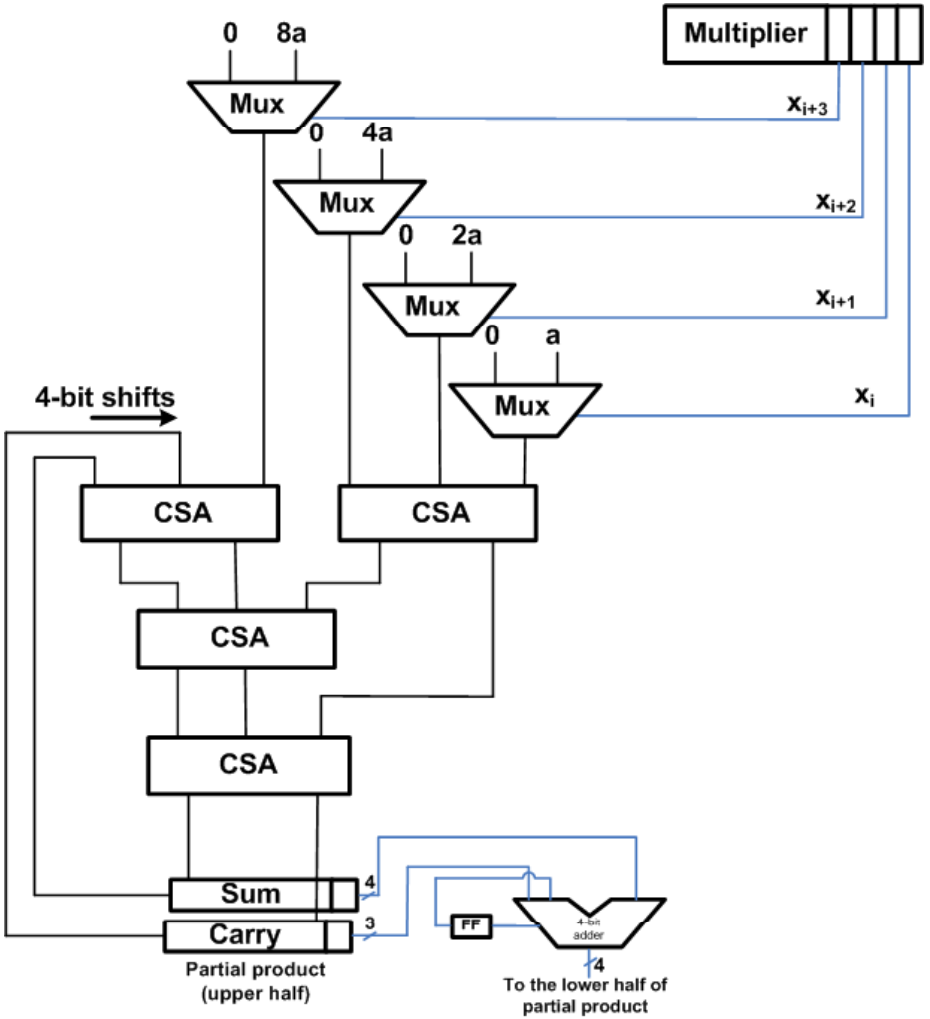
Using Carry-Save Adders

- This is another way to accommodate the required 3a multiple
- Four numbers (the sum and carry components of the cumulative partial products, $x_i a$ and $2x_{i+1}a$) need to be combined, thus necessitating a two-level CSA tree



High-Radix Multipliers

- Now, it is an easy step to visualize a higher-radix multiplier:
  - In radix-2^b multiplication with Booth's recoding, we have to reduce b/2 multiples to 2 using a (b/2+2)-input CSA tree whose other two inputs are taken by the carry-save partial products. Without Booth's recoding a (b+2)-input CSA tree would be needed

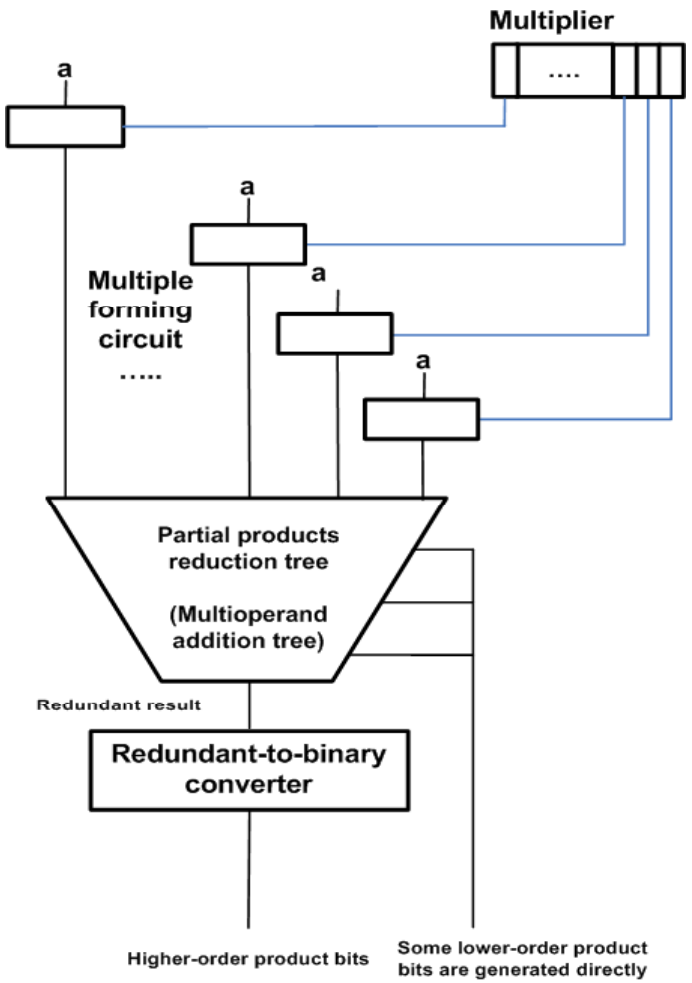


Tree and Array Multipliers

- Tree, or fully parallel multipliers constitute limiting cases of high-radix multipliers (radix-2^k)
- With a high-performance CSA tree followed by a fast adder, logarithmic time multiplication becomes possible
- The resulting multipliers are expensive, but justifiable, for applications in which multiplication speed is critical
- One-sided CSA trees lead to much slower, but highly regular, structures known as array multipliers that offer higher pipelined throughput than tree multipliers and significantly lower chip area

Full-Tree Multipliers

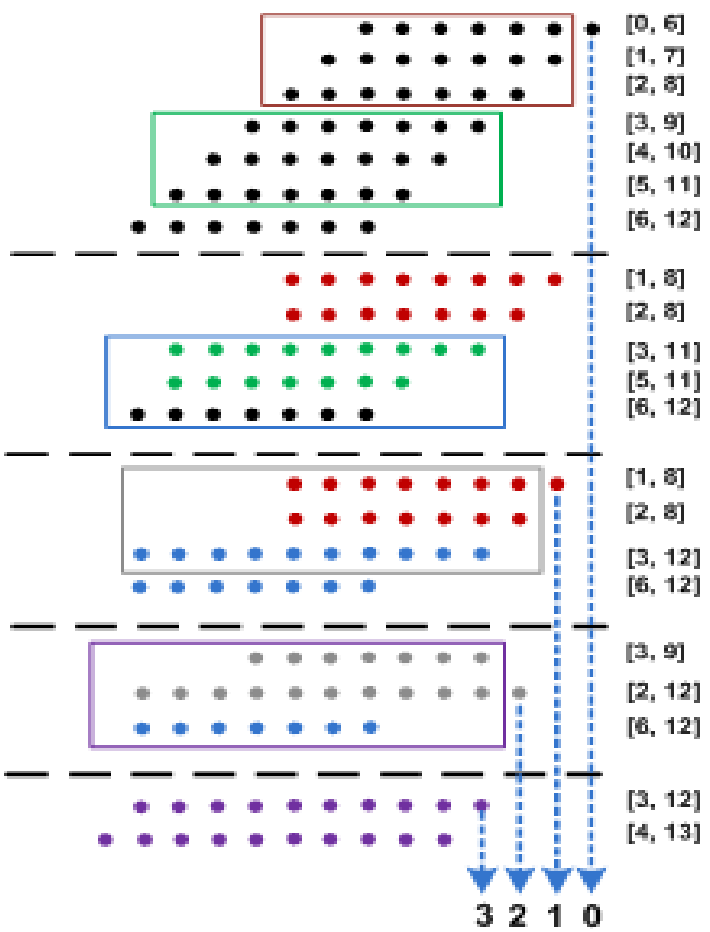
- In full-tree multipliers, all the k multiples of multiplicand are produced at once and a k-input CSA tree is used
- All the multiples are combined in one pass; the tree does not require feedback links, making pipelining quite feasible

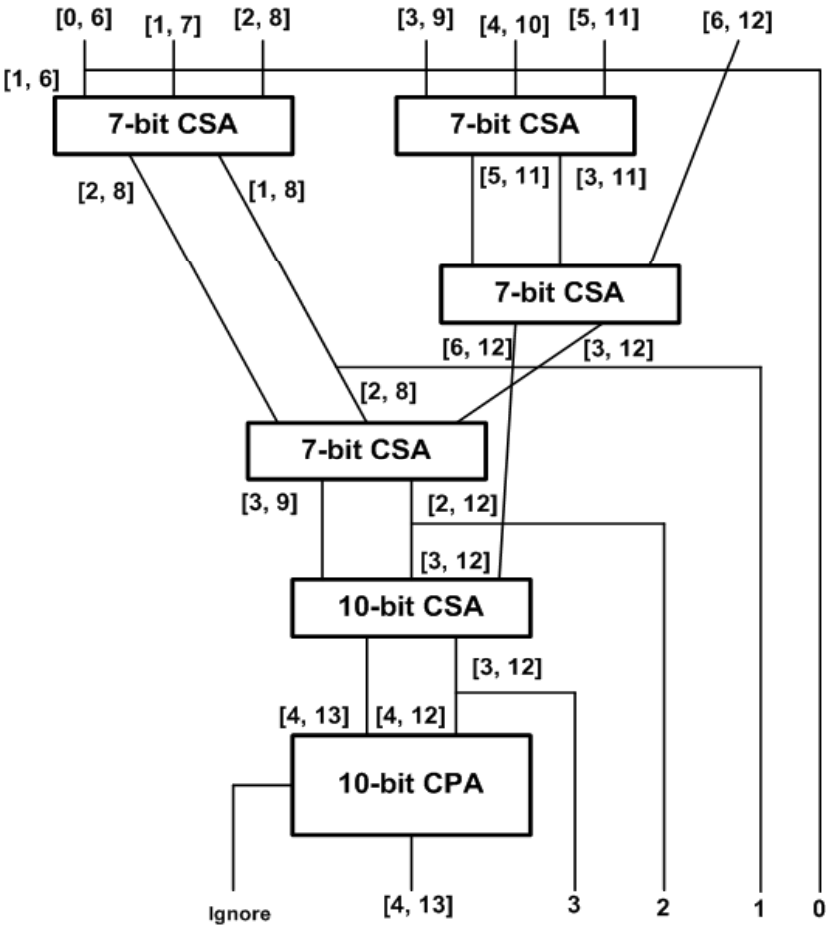


Reduction Tree

- A logarithmic depth reduction tree based on CSA, has an irregular structure that makes its design and layout quite difficult
- Additionally, connections and signal paths of varying lengths lead to logic hazards and signal skew that have implications for both performance and power consumption
- Compared to generic CSA, the only modification required is relative shifting of the operands to be added

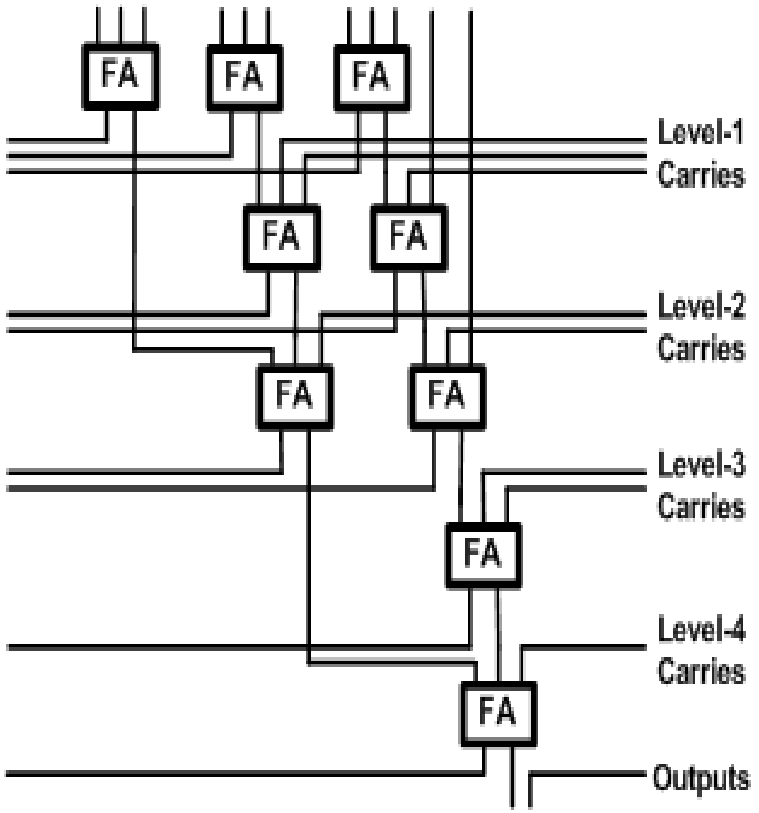
Reduction Tree





Alternative Reduction Trees

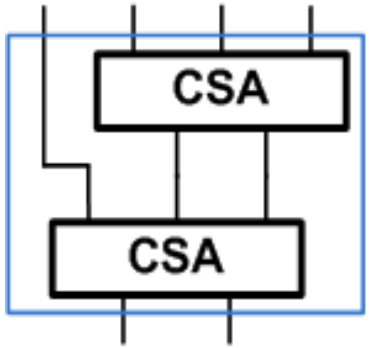
- A slice of (n;2) counter, when suitably replicated, can perform the function of the reduction tree
- Using counters assures us that all outputs are produced after the same number of full-adder delays
- The structure can be replicated to form an n-input reduction tree of desired width. Such balanced-delay trees are quite suitable for VLSI implementation of parallel multipliers

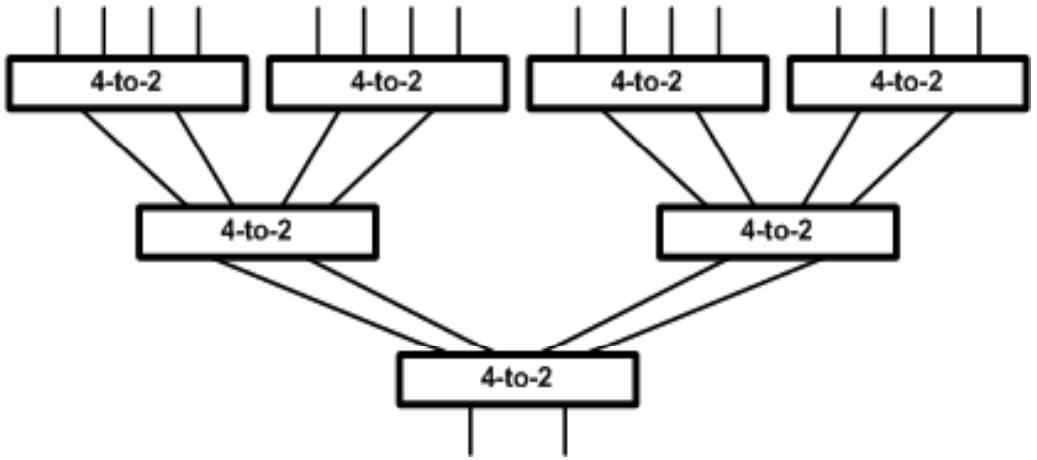


Alternative Reduction Trees

- Another alternative is using a module that reduces four numbers to two as the basic building block

- Then partial products reduction trees can be structured as binary trees that possess a recursive structure, making them more regular and easier to layout



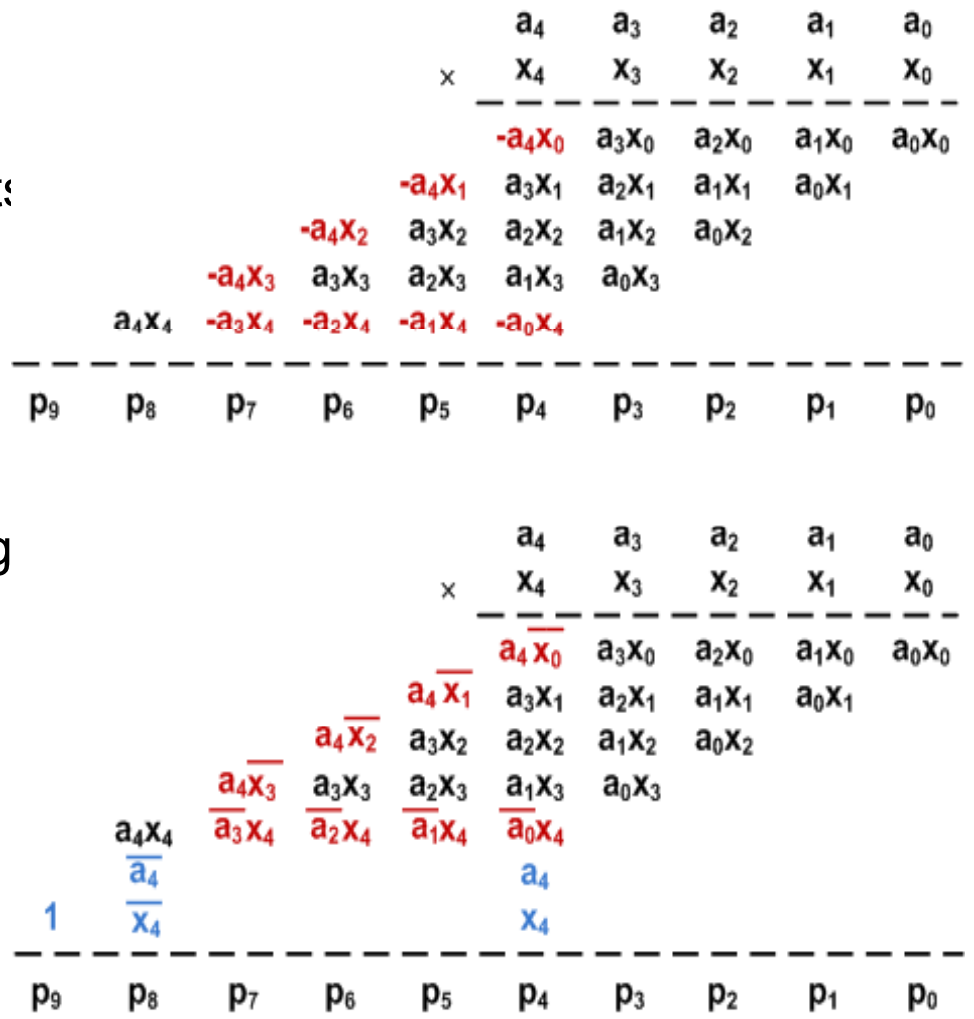


Tree multipliers for signed numbers

- In multiplying 2's-complement numbers directly, partial products are signed numbers

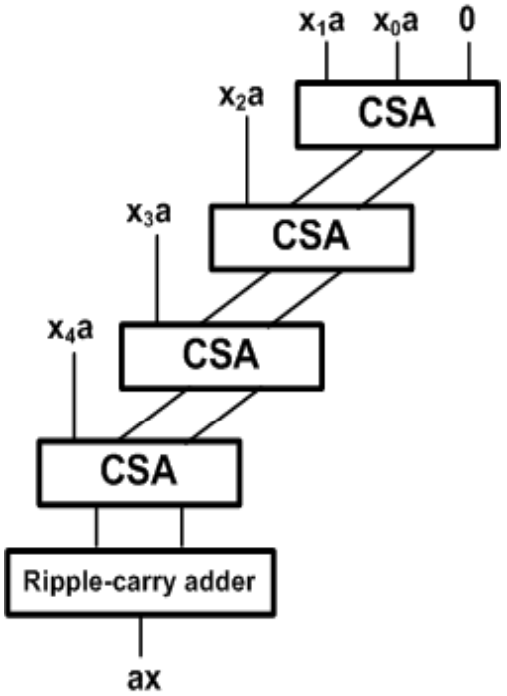
- To avoid having to deal with negatively weighted bits, an efficient method offered by Baug and Wooley:

  $-x_0 = \overline{x}_0 - 1$



Array Multipliers

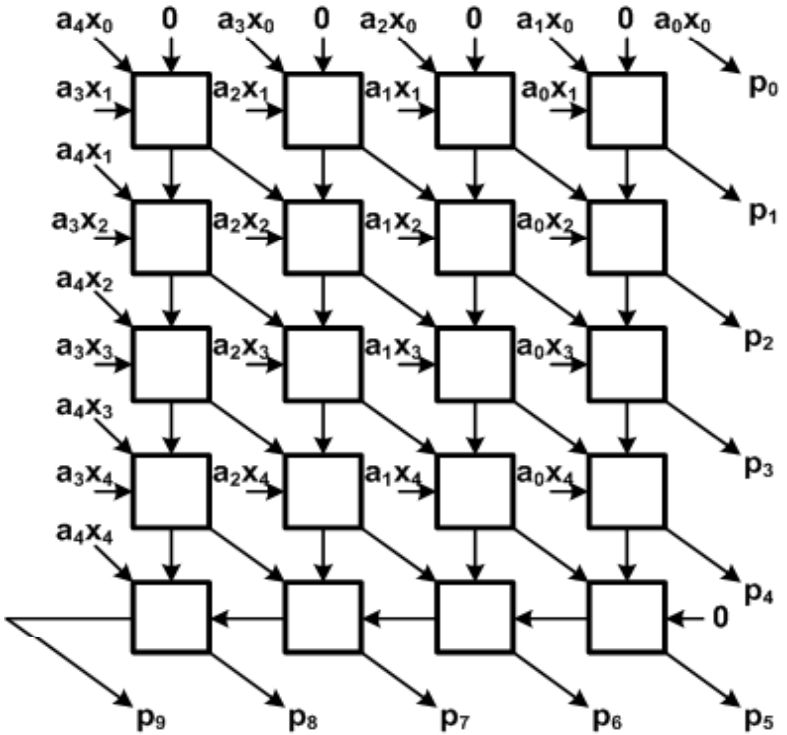
- A tree multiplier, with a one-sided reduction tree and a ripple-carry final adder is called an array multiplier
- an array multiplier is very regular in its structure and uses only short wires that go from one FA to adjacent FA
- It has a very simple and efficient layout in VLSI and can be easily and efficiently pipelined



Array Multipliers

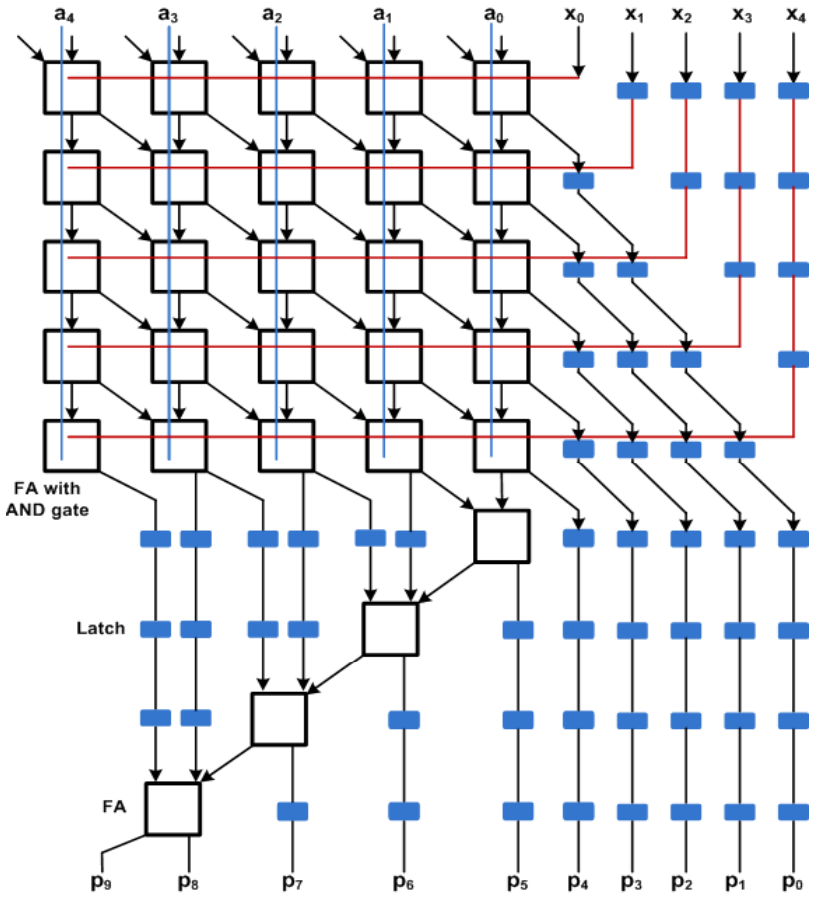
- Sum outputs are connected diagonally, while the carry outputs are linked vertically, except in the last row, where they are chained from right to left

- Baugh and Wooley method can be easily applied to array multiplier for 2's-complement multiplication



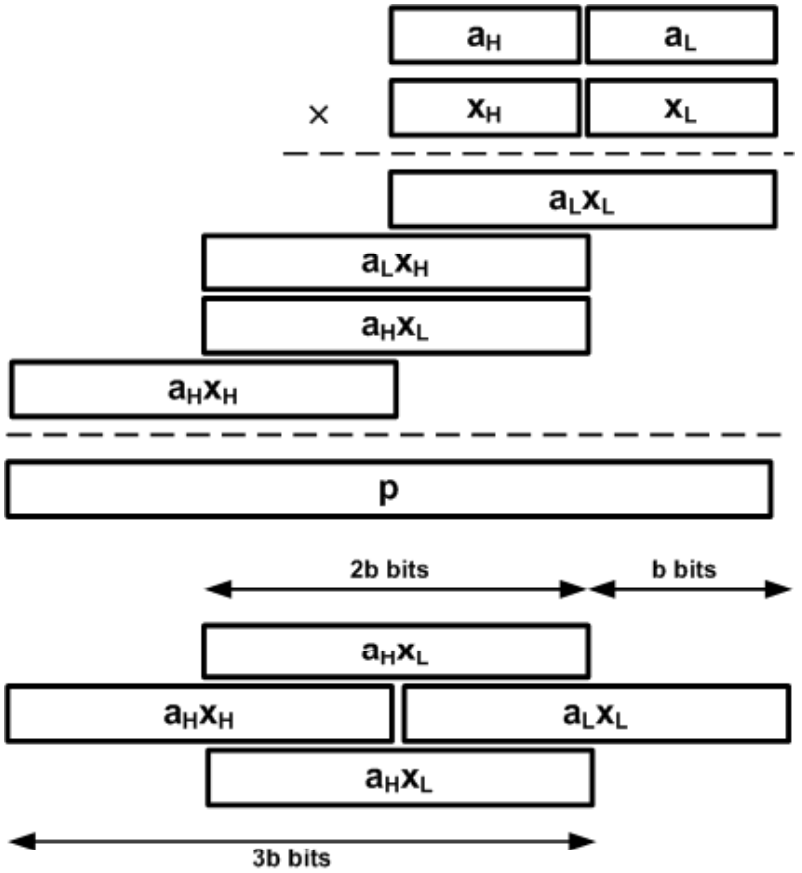
Pipelined Tree and Array Multipliers

- X_i inputs are delayed through the insertion of latches in their paths and the product emerges with a latency of 2k-1 cycles
- FA blocks used are assumed to have output latches for both sum and carry
- The final ripple-carry adder has been pipelined as well



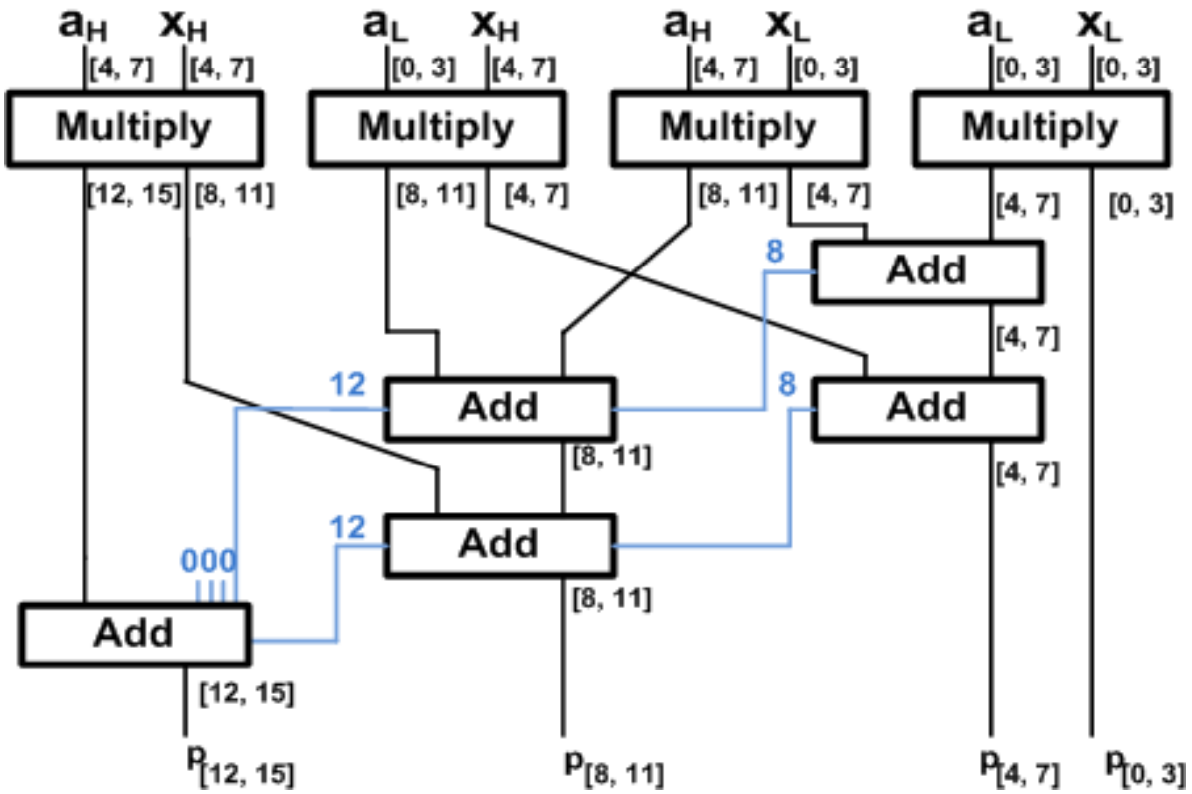
Divide and Conquer Design

- A 2b×2b multiplier can be synthesized using b×b multiplier
- Although there are four partial products, only three values need to be added
- 2b×2b multiplication has been reduced to 4 b×b multiplications and a three-operand addition



Divide and Conquer Design

- For 2b×2b multiplication one can use b-bit adders exclusively to accumulate the partial products



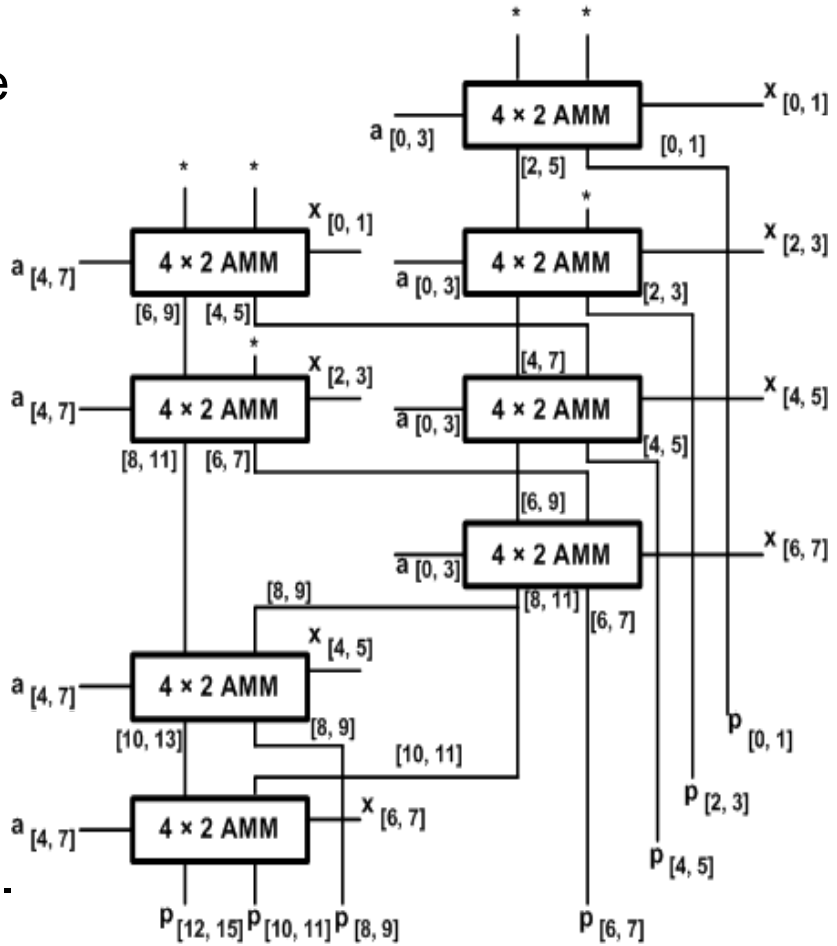
Additive Multiply Modules (AMMs)

- In certain computations, multiplications are commonly followed by additions. In such cases, implementing a multiply-add unit to compute p=ax+y might be cost effective. Furthermore, AMMs can be used as building blocks for multipliers

- In a b×c AMM:

  $(2^b-1)(2^c-1)+(2^b-1)+(2^c-1)=2^{b+c}-1$

- The cost of a 4×2 AMM is less than the combined costs of a 4×2 multiplier and a 4-bit adder

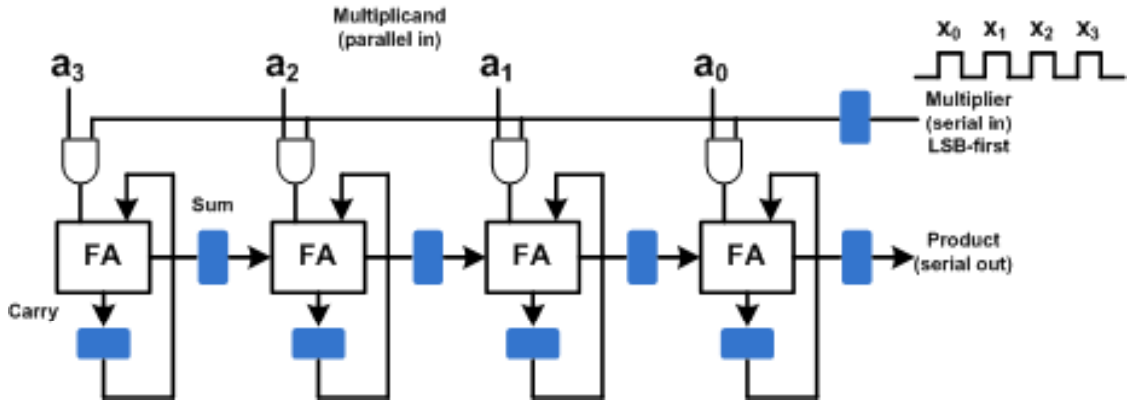


Inputs marked with an asterisk carry 0s

Bit-Serial Multipliers

- Bit-serial arithmetic is attractive in view of its smaller pin count, reduced wire length, and lower floor space requirements in VLSI

- The compactness of the design may allow it to run a bit-serial multiplier at a high enough clock rate to make it competitive with much more complex designs with regard to speed



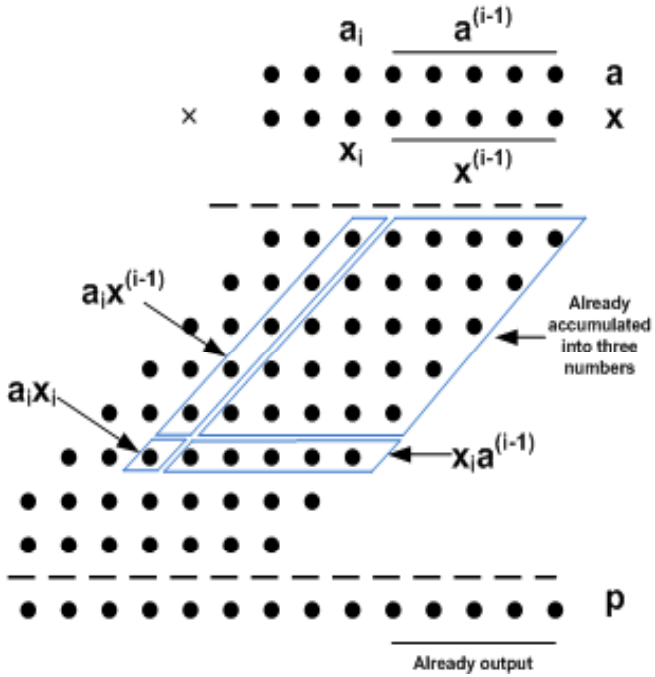
Bit-Serial Multipliers

- For a latency-free multiplier, the relationship between the output and inputs are written in the form of a recurrence:

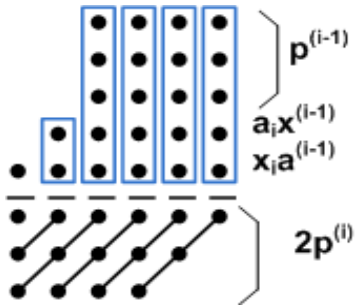
$a^{(0)}=a_0$, $a^{(1)}=(a_1a_0)_2$, … , $a^{(i)}=2^ia_i+a^{(i-1)}$

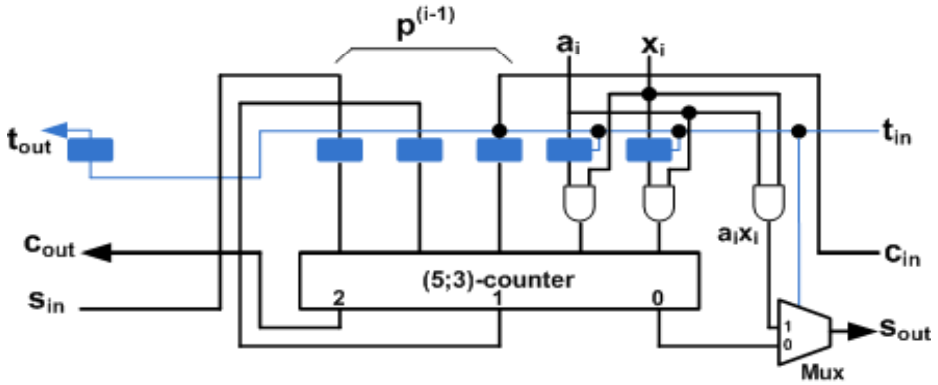
$p^{(i)}=2^{-(i+1)}\, a^{(i)}\, x^{(i)}$,

$2p^{(i)}=p^{(i-1)}+a_ix^{(i-1)}+x_ia^{(i-1)}+2^ia_ix_i$



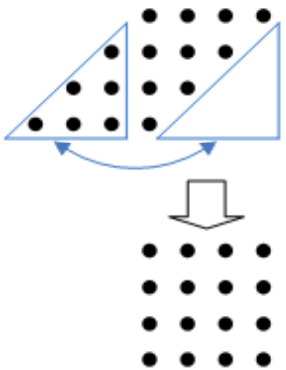
- A (5;3) counter can be used as an adder, if $p^{(i-1)}$ is stored in double-carry-save form

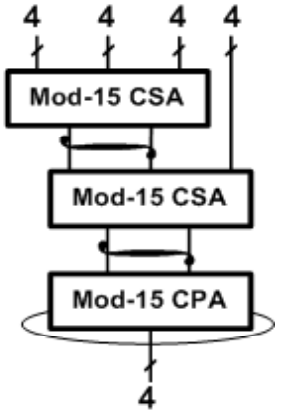




Modular Multipliers

- A modular multiplier is one that produces the product of two (unsigned) integers modulo some fixed constant m. The two special cases of $m=2^b$ and $m=2^b-1$ are simpler to deal with
- If the partial products are accumulated through carry-save addition,
  - for $m=2^b$, the output carry in position b-1 is ignored
  - for $m=2^b-1$, the carry out of position b-1 is combined with bits in column 0





Modular Multipliers

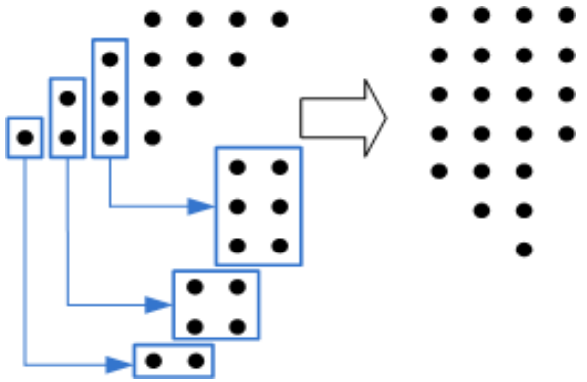
- Similar techniques can be used to handle modular multiplication in the general case

- As an example, a modulo-13 multiplier can be designed by using identities:

  $16=3 \bmod 13 \quad 3 \rightarrow 2+1$

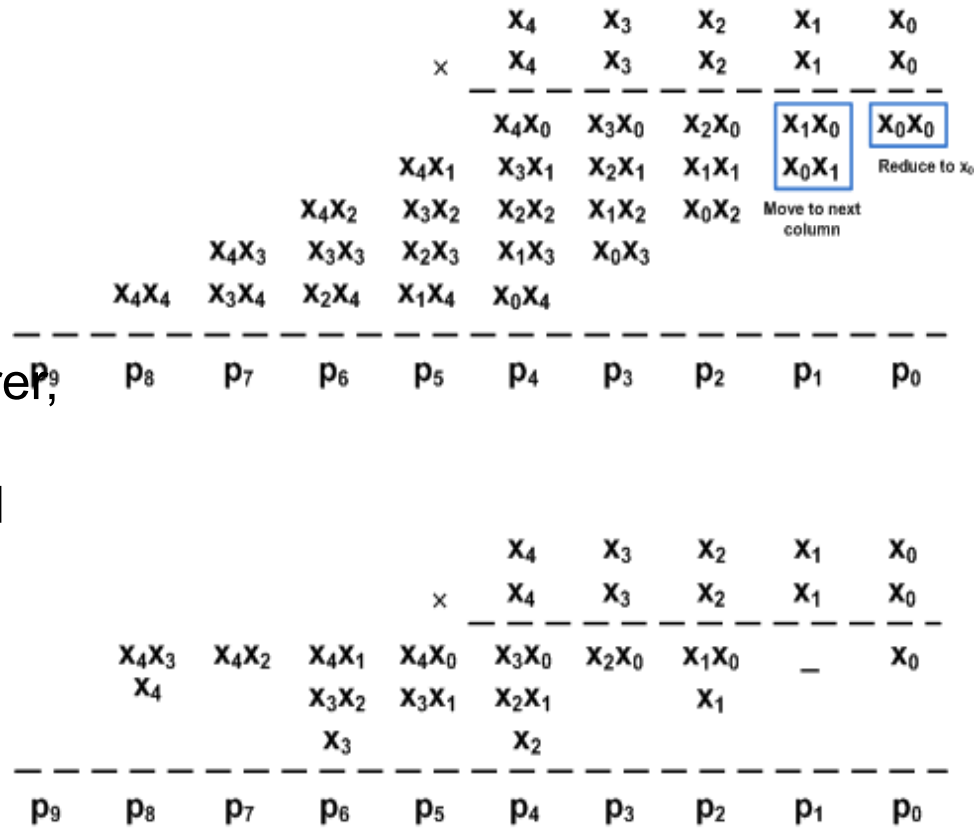
  $32=6 \bmod 13 \quad 6 \rightarrow 4+2$

  $64=12 \bmod 13 \quad 12 \rightarrow 8+4$



Squaring

- Any standard or modular multiplier can be used for computing $p=x^2$ if both inputs are connected to x
- A special-purpose k-bit squarer, if built in hardware, will be significantly lower in cost and delay than a k×k multiplier
  - $x_i x_i \rightarrow x_i$
  - $x_i x_j + x_j x_i \rightarrow 2x_i x_j$



Conclusion

- The classic shift/add multiplication schemes and their implementation have been examined
- There are two ways to speed up the underlying multi-operand addition; reducing the number of operands leads to high-radix multipliers, and devising hardware multi-operand adders that minimize the latency and/or maximize the throughput leads to tree and array multipliers
- Cost, VLSI area, and pin limitations favor bit-serial designs, while the desire to use available building blocks leads to designs based on Additive Multiply Modules (AMMs)
- Finally, the special case of squaring was of interest, as it leads to considerable simplification

Questions and Comments